Lesson 9

Priority Queues:
Rapid Access on the Ground of Orderliness

Wholeness of the Lesson

Queues provide rapid insertions and removal of elements, adhering to the rule: "first in, first out." In a Priority Queue, one is allowed to remove elements in any desired order (not based on when the element was inserted). Implementing this idea using the heap data structure results in O(log n) performance of the main operations.

Science of Consciousness: Pure consciousness is the field of pure orderliness and the field of all possibilities. Experience of this field shows that, on the ground of the orderliness experienced from transcending, new possibilities arise in daily life to open new directions and solve problems.

Main Idea Behind Priority Queues

◆ In an ordinary queue, elements are removed from the queue in the order in which they were originally inserted.

◆ Sometimes, a different order of removal would be preferable. Example: In an operating system, a time slice is given to a requesting process according to priority rather than according to the order in which processes place requests.

◆ Priority Queues support the notion that highest priority objects are removed first.

Priority Queue ADT

◆ *Elements of form (k,e).* In order to specify priorities, each object to be inserted into the queue is equipped with a key – its "priority". Therefore, typical elements of the queue are pairs (k,e) where k is the key (like an integer) and e is the element. (Duplicate keys are allowed.)

◆ *Operations.* A Priority Queue has two main operations:

  `insertItem(k,e)` – inserts the pair (k,e)

  `removeMin()` – removes a pair (k,e) for which k is smallest and returns e

  Auxiliary Operations:

  `minElement()` – returns (but does not remove) an element with smallest key

  `minKey()` – returns (but does not remove) the smallest key

Implementations of Priority Queues

◆ *Background data structure is an unsorted array.*
- `removeMin` takes $\Theta(n)$ on average
- insertItem takes $\Theta(1)$ if no reconstruction of the array needed

◆ *Background data structure is a sorted array*
- `removeMin` takes $\Theta(1)$ if implemented properly (could arrange to have array in reverse sorted order so the min is always last element)
- `insertItem` takes $\Theta(n)$ on average, with many copies of array elements

◆ *Background data structure is a red-black tree*
- `removeMin` takes $O(\log n)$ – remove the min, located at the end of leftmost branch ($O(\log n)$)
- `insertItem` takes $O(\log n)$ – this is just red-black tree insertion

◆ Red-black tree performance is nearly optimal, but can be improved slightly using a different approach.

Heaps

- A heap is a binary tree that has a structural property and an ordering property.

    - *Heap Structural Property:* Structurally, a heap is a binary tree in which every level except possibly the bottom level is filled completely, and the bottom level is filled from left to right. (Such trees are sometimes called *complete.*)

    - *Min-Heap Order Property:* A key in one node n must always be greater than or equal to the key in the parent of n (if there is a parent).

        <u>Note</u>: The Min-Heap Order Property ensures min key is at the root. A variant is the *Max-Heap Order Property* (a key in one node n must always be less than or equal to the parent key) ensures the max key is at the root.

Heaps

- The structural property implies that if the heap has height h with n nodes, then

$$2^h \le n \le 2^{h+1} - 1$$

  Taking logs establishes:

  Theorem. If T is a heap with n nodes and height h, then h is O(log n) (with small constant factor).

- Consequently, when a heap is used to implement a PriorityQueue, removeMin and insertItem run in O(log n) and the constant factors are smaller than if a Red-Black tree is used.

`insertItem` Using Heaps

◆ Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap

◆ The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)
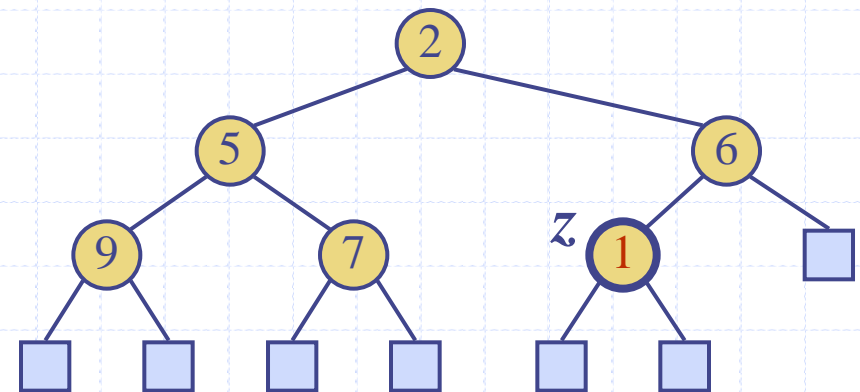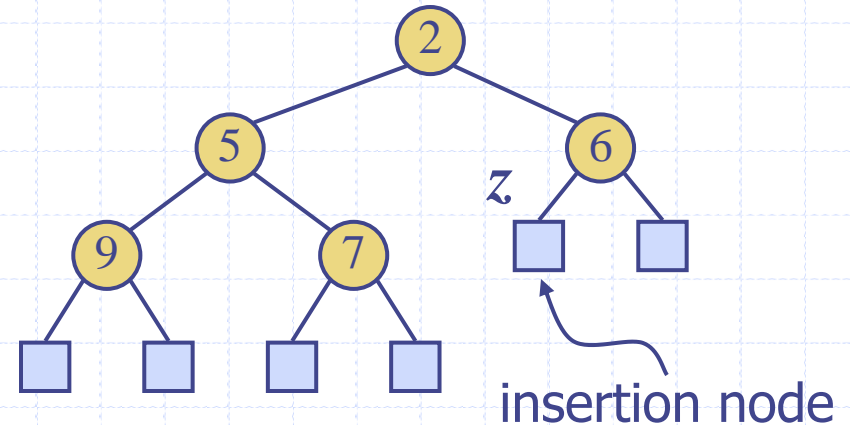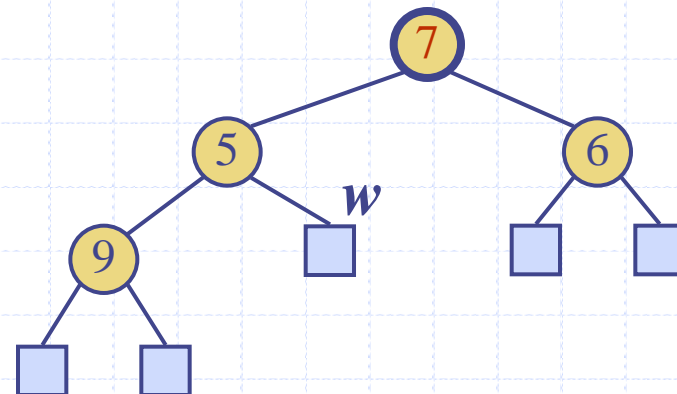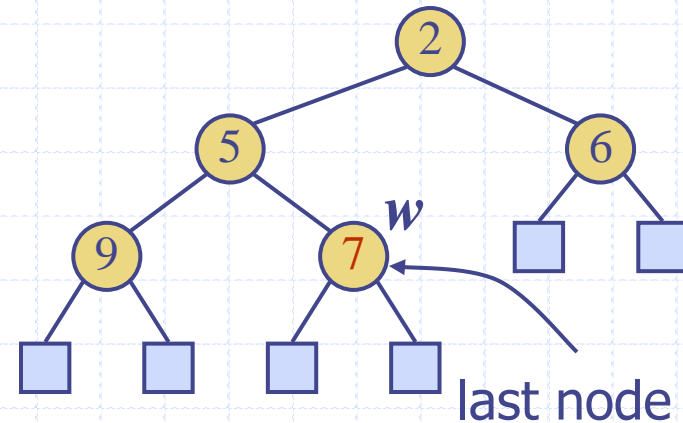
insertion node

Maintaining Heap Order: `Upheap`

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
- NOTE: During upheap, a swap occurs only if the key k that is being moved upward is *strictly less than* its parent.
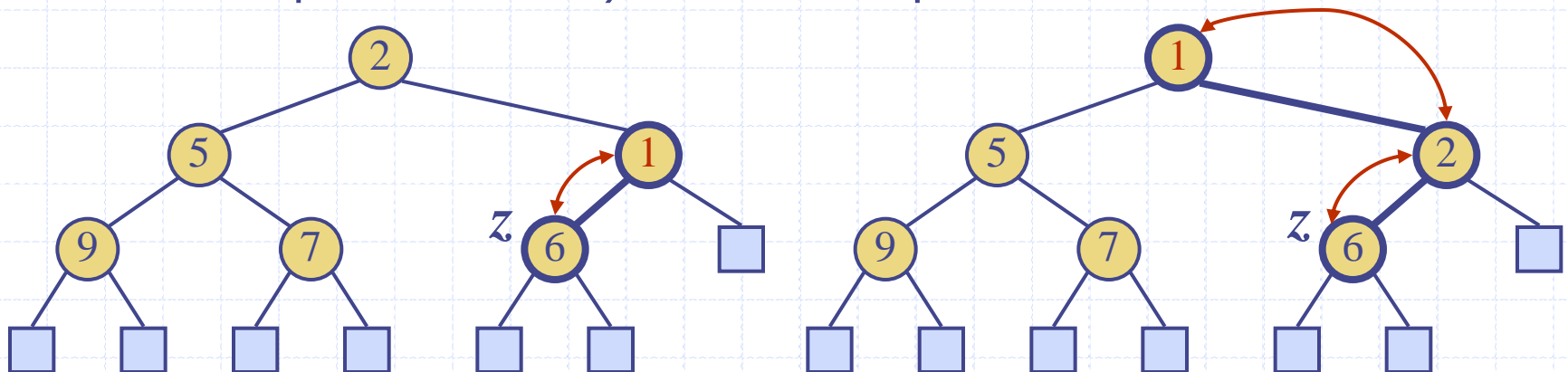
`removeMin` Using Heaps

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap

- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Make the last node null
  - Restore the heap-order property (discussed next)

last node

Maintaining Heap Order after Removal: `Downheap`

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time
- NOTE: In the downward path, whenever key k is larger than both the child values, k is swapped with the *smaller* of the two values.
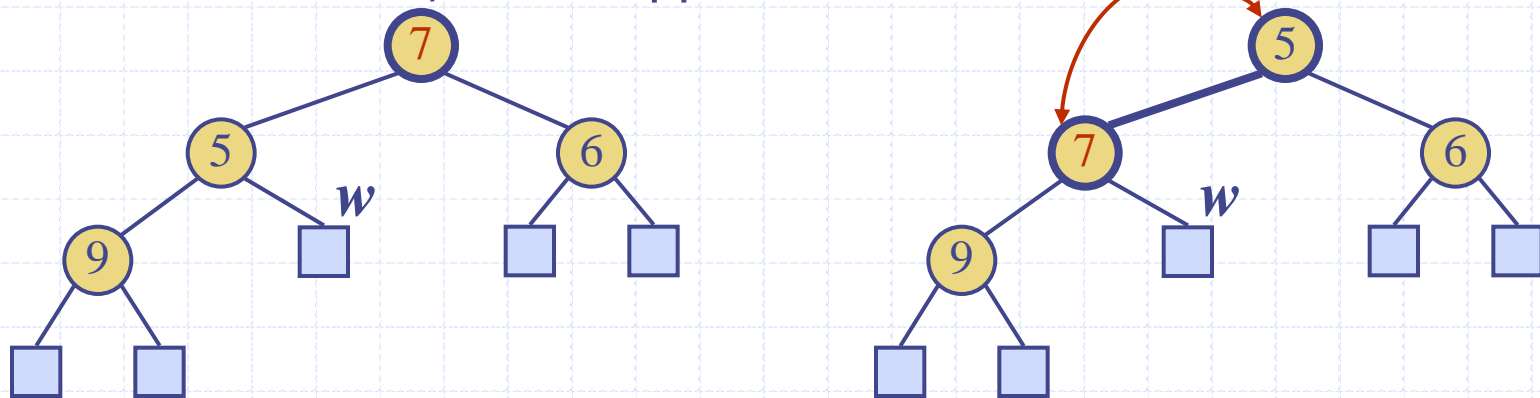
Implementing a Heap in an Array

- We can represent a heap with n keys by means of an array of length n

- For the node at position i
  - the left child is at position $2i+1$
  - the right child is at position $2i + 2$
  - the parent is at position $(i-1)/2$.

- Operation insertItem corresponds to inserting at next available slot

- Operation removeMin corresponds to removing item in first occupied slot

2	5	6	9	7
0	1	2	3	4
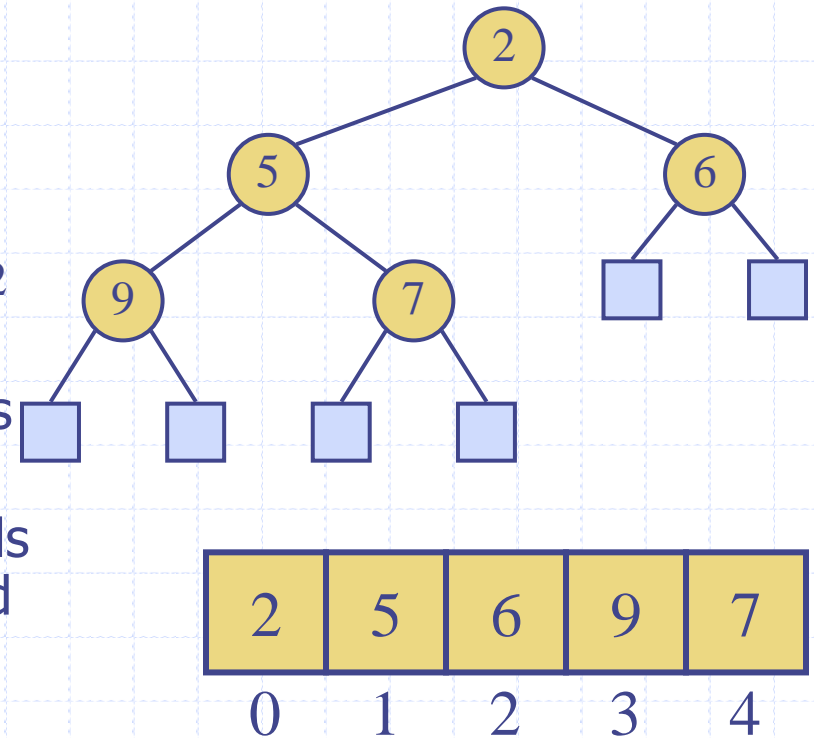
Main Point

The most efficient implementation of a Priority Queue is via *heaps* which support $O(\log n)$ worst-case running time for both operations `removeMin` and `insertItem`, with minimal overhead. A heap is a binary tree in which every level is filled except possibly the bottom level, which is as full as possible from left to right. In addition, a binary heap satisfies the *heap order property:* For every node *X*, the key value of *X* is greater than or equal to that of its parent (if it has a parent). A Priority Queue gives a simple model of the principle of the Highest First: Putting attention on the highest value as the top priority results in fulfillment of all lower-priority values as well, in the proper time.

Using a Heap for Sorting

◆ Using a heap for sorting: Given an array of *n* items (keys) to sort:

 ❖ Use `insertItem` *n* times to load the heap

 ❖ Output the results by inserting into output array the output of `removeMin`, loading from left to right

Algorithm HeapSort(S)
   Input: Sequence S of n elements
   Output: S in sorted order
   P ← a MinHeap
  while !S.isEmpty() **do**
     e ← S.remove(S.first())
     P.insertItem(e)

  while !P.isEmpty() **do**
     e ←P.removeMin()
     S.insertLast(e)
  return S

Using a Heap for Sorting

- Running time:
 - load the heap: O(nlog n)
  - load the output array: O(nlog n)

  Therefore *HeapSort* runs in O(nlog n), in worst-case

- Demo: Compare HeapSort with other O(nlog n) sorting algorithms

HeapSort Optimization: *Bottom-Up Construction*

◆ We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases

◆ In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
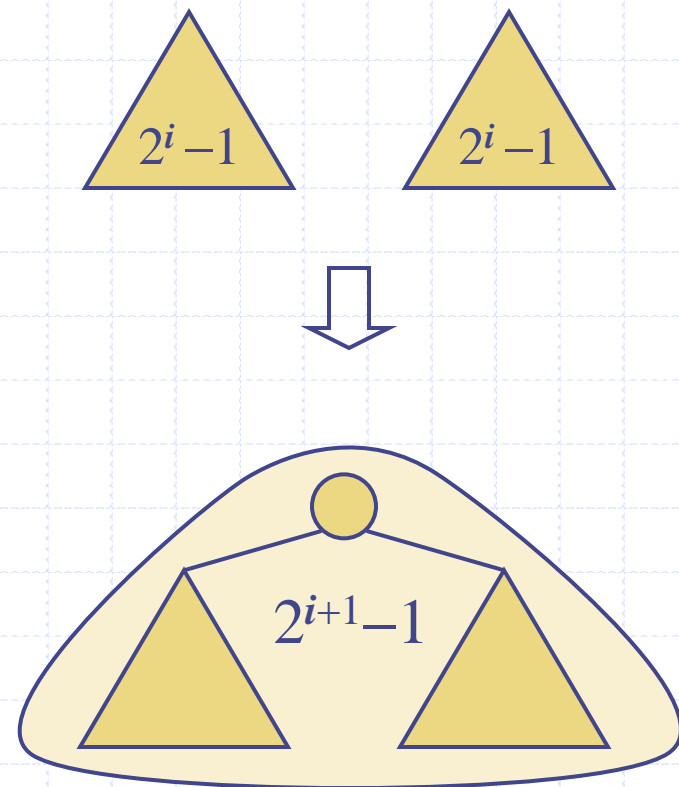
(Continued)

Algorithm BottomUpHeap(A)

    Input: An array A storing n = 2^h-1 keys

    Output: A heap T storing the keys in A

    if $|A| \leq 1$ **then**

       return A

    remove the first key *k* from A

    split A into two subarrays, A_1 and A_2, of equal length

    $T_1 \leftarrow$ BottomUpHeap(A_1)

    $T_2 \leftarrow$ BottomUpHeap(A_2)

    create a binary tree T with root r storing k, left subtree T_1, right subtree T_2, perform a downheap from the root r of T, if necessary

    return T

NOTE: The algorithm works without modification when n has the form 2^h-1; it needs to be modified slightly for other n (but the results concerning its running time still hold true).

In-Class Exercise

Do recursive Bottom-Up-Heap on the input [4, 5, 2, 6, 4, 7, 3]

Optional: A Computation of BottomUpHeap Running Time

◆ An examination of BottomUpHeap recursive algorithm suggests that its running time satisfies the following recurrence:

$$T(1) = d$$

$$T(n) = 2T(n/2) + \log n \quad (n \text{ a power of } 2)$$

(Continued)

Using the Guessing Method, we obtain a solution to the recurrence as follows:

$$T(1) = d$$

$$T(2) = 2d + 1$$

$$T(4) = 2(2d + 1) + 2 = 2^2 \cdot d + 2^1 \cdot 1 + 2$$

$$T(8) = 2(2^2 \cdot d + 2^1 \cdot 1 + 2) + 3 = 2^3 \cdot d + 2^2 \cdot 1 + 2^1 \cdot 2 + 3$$

$$T(16) = 2(2^3 \cdot d + 2^2 \cdot 1 + 2^1 \cdot 2 + 3) + 4 = 2^4 \cdot d + 2^3 \cdot 1 + 2^2 \cdot 2 + 2^1 \cdot 3 + 4$$

$$\bullet$$

$$\bullet$$

$$\bullet$$

$$T(2^m) = 2^m \cdot d + \sum_{i=0}^{m-1} 2^i (m - i)$$

We show that the sum $\sum_{i=0}^{m-1} 2^i (m - i)$ is $O(2^m)$ — this suffices to establish the result.

(Contiued)

We show that

$$\lim_{m \to \infty} \frac{\sum_{i=0}^{m-1} 2^i (m-i)}{2^m} \quad \text{is finite.}$$

But

$$\frac{\sum_{i=0}^{m-1} 2^i (m-i)}{2^m} = \frac{m}{2^m} + \frac{m-1}{2^{m-1}} + \frac{m-2}{2^{m-2}} + \ldots + \frac{1}{2} = \sum_{i=1}^{m} \frac{i}{2^i}$$

Recall that

$$\lim_{m \to \infty} \sum_{i=1}^{m} \frac{i}{2^i} = \sum_{i=1}^{\infty} \frac{i}{2^i} = 2,$$

as required.

BottomUpHeap Iteratively

◈ Basic idea is to build the heap from the bottom up

◈ Iterative strategy:

- Make (n+1)/2 single item heap trees--(n-1)/2 items still in reserve.
- Merge pairs of single item heaps by adding an item from the reserve. (n+1)/4 heaps now and do downheap as necessary
- Merge pairs of three item heaps by adding item from the reserve, followed by downheap
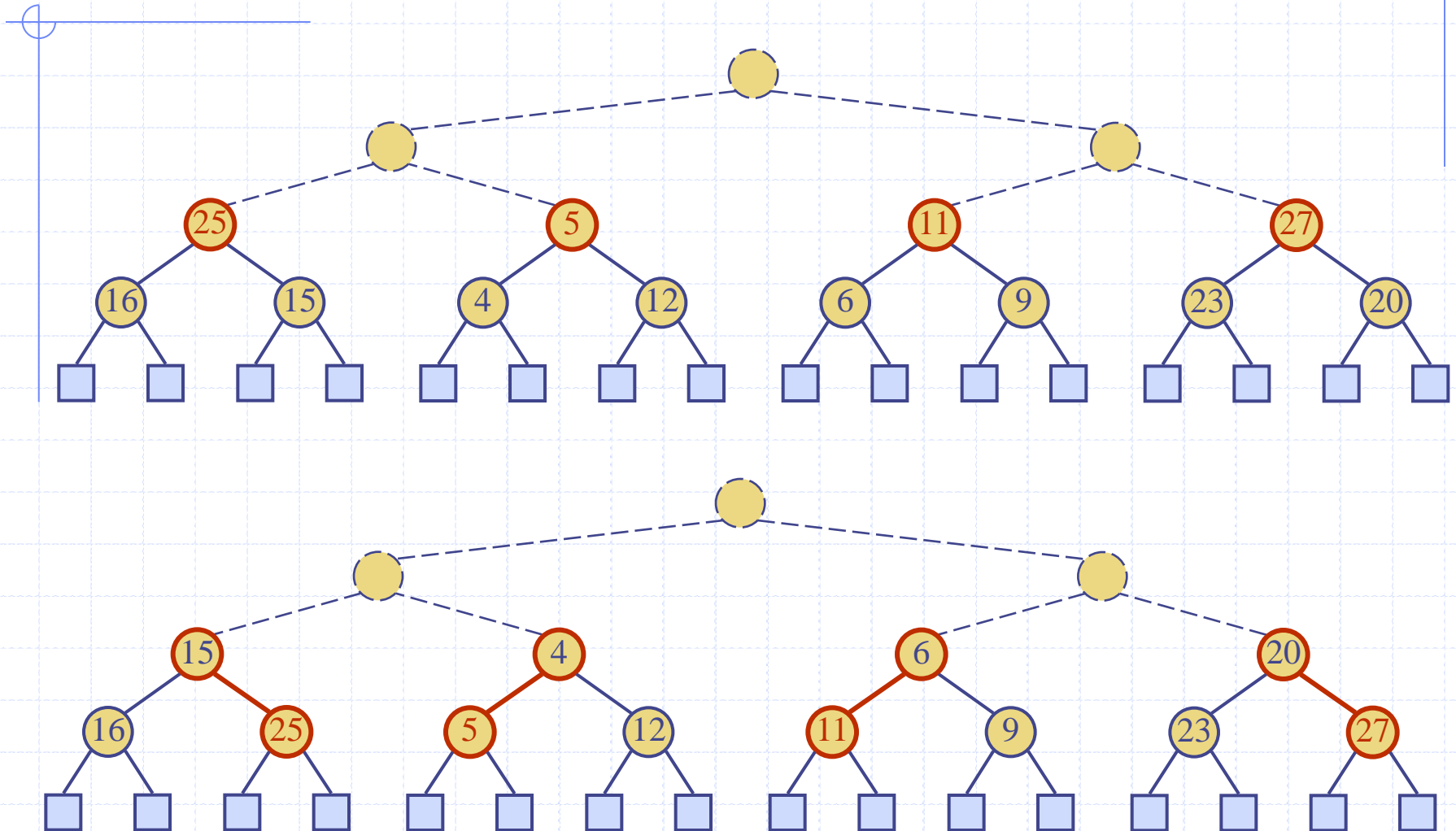- Repeat log(n+1) times (= height of the tree)

Example

A = [16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10]
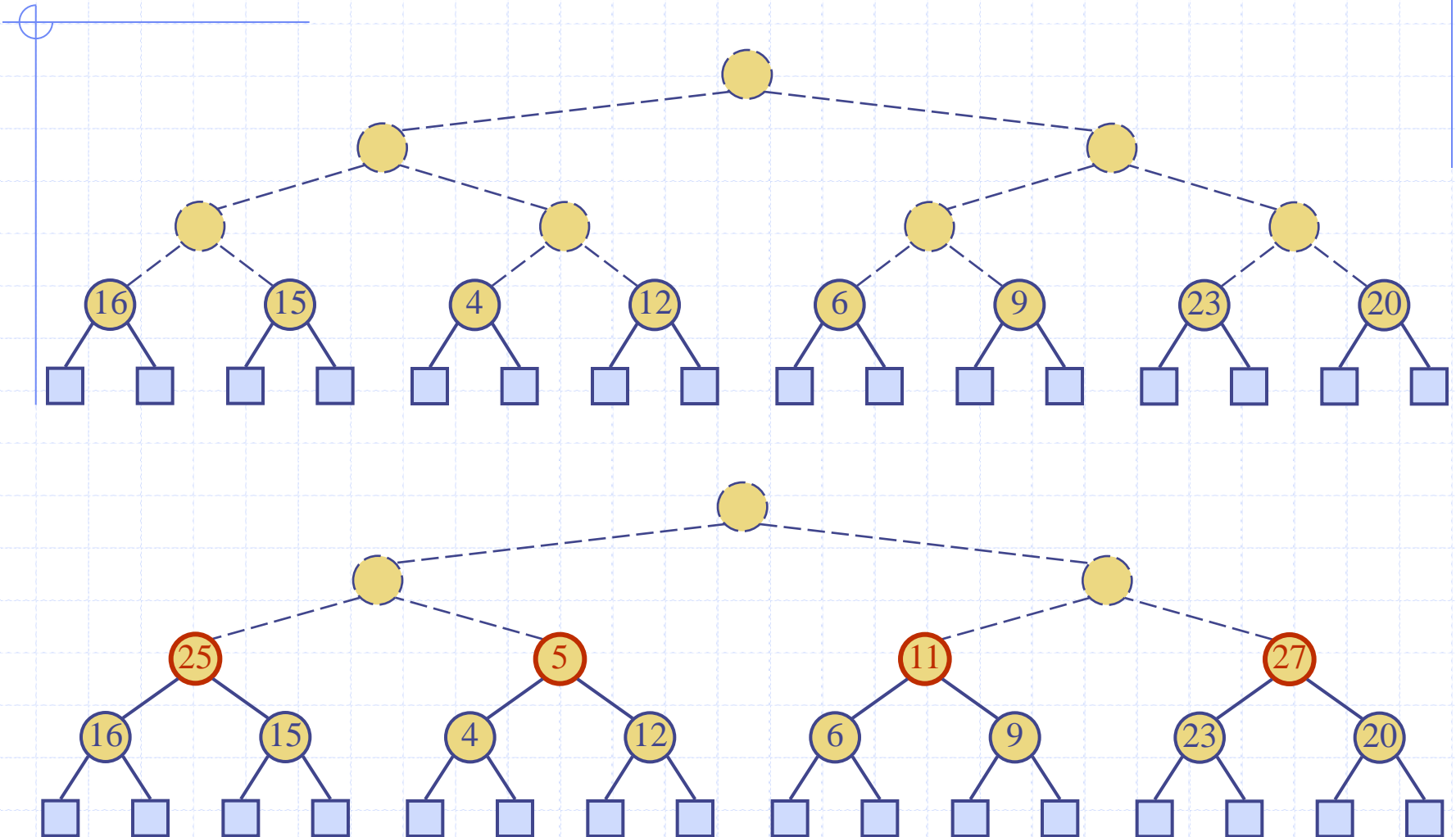
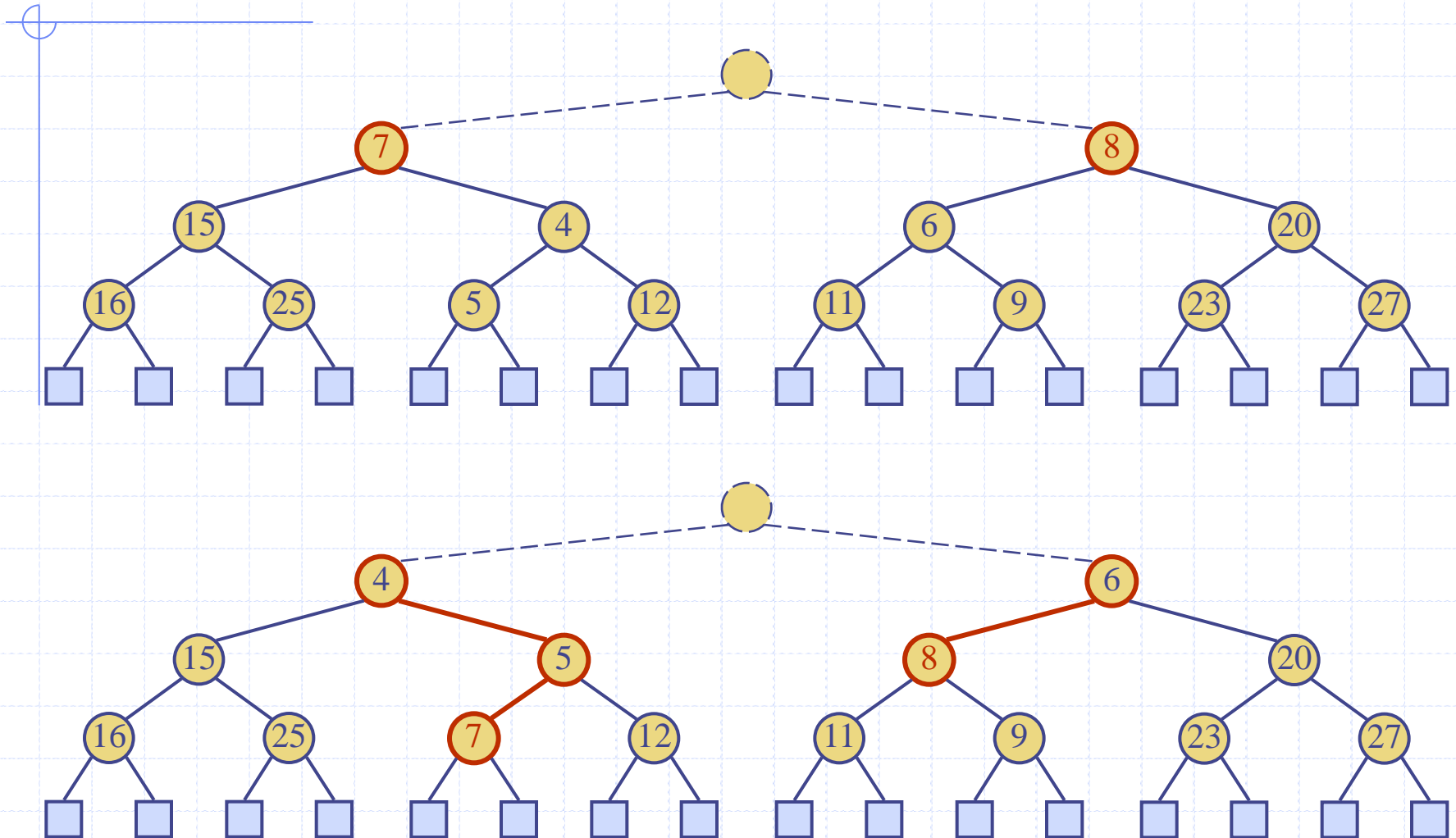# Example (contd.)

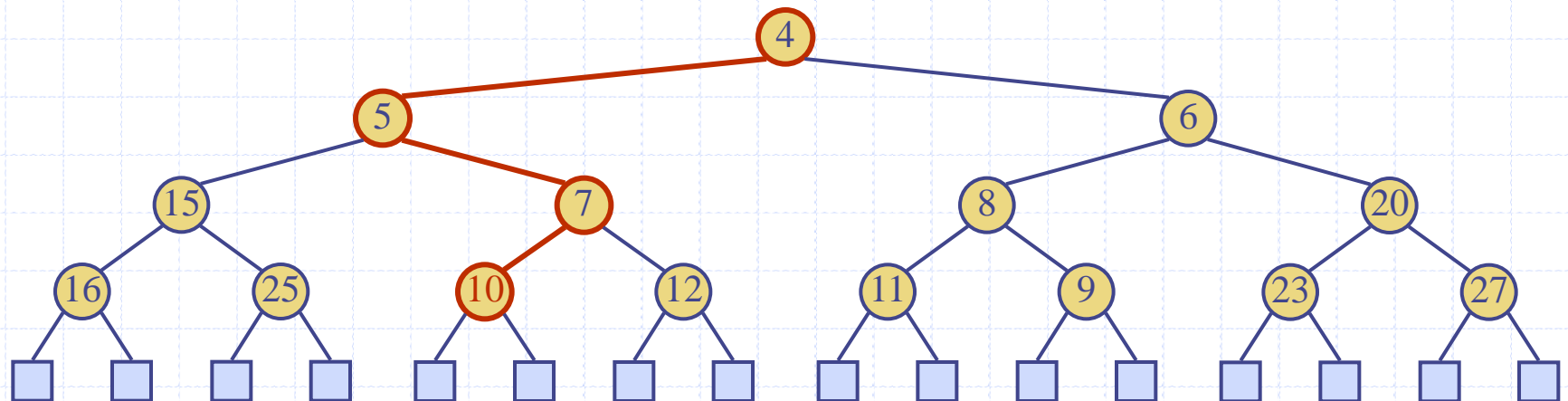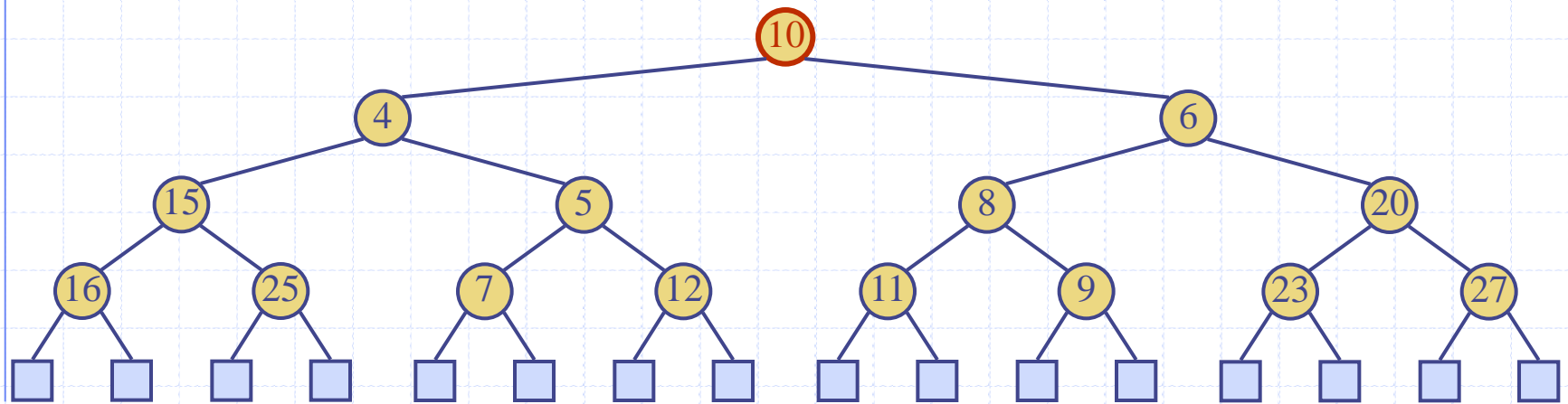A = [16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10]

Example (contd.)

A = [16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10]

Example (contd.)

A = [16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10]

Optional: Proof of O(n) Running Time for Iterative BUH

We assume $n = 2^{h+1} - 1$.

- In the first step, 2^h nodes are laid out, and no downheap operations are required

- In the second step, $2^{h-1} = 2^h/2$ nodes are used as roots of size-3 heaps; each such heap may require $1 = \log 2$ downheap operation

- In the third step, 2^{h-2} nodes are used as roots of size-7 heaps; each such heap may require $2 = \log 2^2$ downheap steps
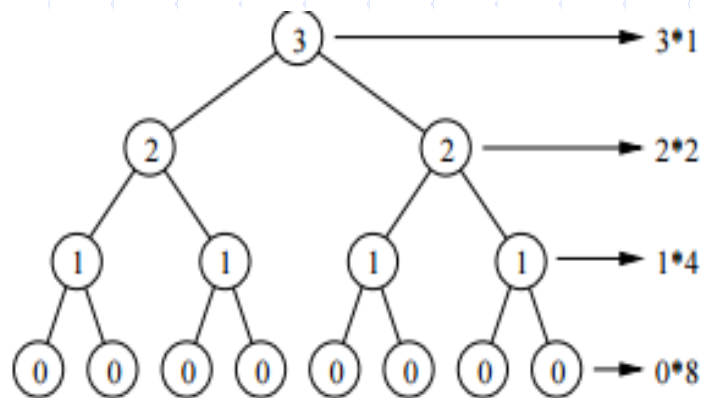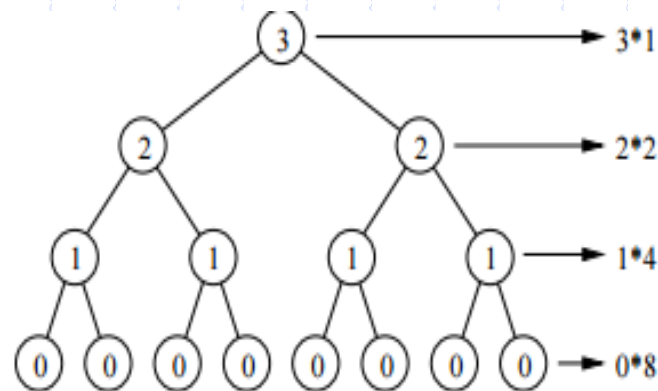
Optional: Proof of O(n) Running Time for Iterative BUH

This sequence 2^h, 2^{h-1}, 2^{h-2}, . . ., 1 has length $1+h$. We form the sum:

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}.$$

Factoring out 2^h, we obtain

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \quad = \; 2*2^h = \; 2^{h+1} = \text{n +1 in O(n)}$$

Optional: Iterative Bottom-up Heap for Any Input Size

- A BUH-number is a number of the form 2^h-1
- The BUH bounds for any positive integer n are consecutive BUH numbers $a < b$ such that $a < n \leq b$.
- Given array A of length n, compute the BUH bounds a, b for n.
- Let $m = b - n$ and $k = n - a$.
- Build lowest level of heap by using first k elements of A
- First step in building second level of heap: get next $\lceil k/2 \rceil$ elements from A to form size-3 heaps with bottom level values (if k is odd, last heap among these will be of size 2 only), then do downheaps for each to ensure order property
- Second setp in building second level of heap: get next $\lfloor m/2 \rfloor$ elements from A to be 1-element heaps at second level
- Next level of heap: Let u be num elements of 2nd level (it will be a power of 2). Take $u/2$ more from A to provide roots for each of the heaps so far.
- Repeat until final element of A is positioned at root of one large heap.

Conclusions

◆ BottomUpHeap loads an array of *n* elements into a heap in O(n) time -- this is faster than the O(nlog n) time of inserting one-by-one. Therefore BottomUpHeap is often used when creating an optimized HeapSort implementation.

◆ As of j2se5.0, Sun provides a PriorityQueue implementation in java.util, using the heap construction.

Main Point

The Bottom Up Heap construction significantly improves the efficiency of building a heap from the usual construction in which one element is added to the heap at a time, with the resulting log n upheap operation every time. Bottom Up Heap proceeds by dividing the input list into three: a root r and two remaining halves. These remaining halves are recursively organized into separate heaps, and then joined together to form the final heap with root r. This process provides an analogy for the process of unfoldment of creation from the unmanifest. In this process, three (rishi, devata, chhandas) emerge from one and then, through self-referral dynamics (a kind of "recursion"), unfold sequentially to form the blueprint of the universe.

Connecting the Parts of Knowledge
With The Wholeness of Knowledge
BOTTOM UP HEAP CONSTRUCTION

1. HeapSort is a fast sorting algorithm that proceeds by first building a heap from an input list and then repeatedly removing the max value to form final sorted output. Each step requires nlog n steps, asymptotically.

2. The Bottom Up Heap construction makes it possible to perform the first phase of HeapSort – the heap-building phase – in O(n) time. This is accomplished by dividing the input list into three, recursively building heaps in two of the three parts, and joining the parts together, returning the final heap.

3. *Transcendental Consciousness* is the unmanifest field of pure consciousness, having a 3-in-1 structure, consisting of rishi, devata, and chhandas.

4. *Impulses within the Transcendental field*: Within the transcendental field, its fundamental 3-in-1 structure gives rise to a self-referral unfoldment of the blueprint of creation, the *Ved.*

5. *Wholeness moving within itself:* In Unity Consciousness, every aspect of experience is appreciated in terms of the 3-in-1 dynamics of consciousness.